Ceng213 - Data Structures
Programming Assignment 2
Inverted Index via Binary Search Trees

METU - Fall 2019

1 Objectives

In this assignment, you are first expected to implement a *binary search tree* data structure. In this binary search tree, each node will contain the data and have two pointers to the root nodes of its left and right subtrees. The data structure will include a single pointer that points to the root node of the binary search tree. The details of the structure are explained further in the following sections. Then, you will use this specialized binary search tree structure to represent *inverted indexes* and implement some functions on them.

Keywords: *Data Structures, Binary Search Trees, Information Retrieval, Inverted Indexes*

2 Inverted Index (Background)[1]

An ***inverted index*** is an index storing a mapping from content, such as words or numbers, to its locations in a document or a set of documents. The purpose of an inverted index is to allow fast text searches. It is the most popular data structure used in document retrieval systems and in search engines like Google.

The inverted index data structure is a central component of a typical search engine indexing algorithm. The goal of a search engine is to optimize the speed of the query: "Find the documents where word X occurs". With the inverted index created, the query can be resolved by jumping to the word X in the inverted index and getting the list of documents where word X occurs.

Inverted index stores a list of occurrences (i.e. document ids) for each word, ***typically in the form of a hash table or a binary search tree***. Many search engines use an inverted index when evaluating a search query to quickly locate documents containing the words in the query and then rank these documents by relevance. As an example, assume there are two documents D1: "my name is john" and D2: "what is your name". The inverted index for these two documents is given in Table 1. To resolve a search query, say "name", the search engine will look up the word "name" in the inverted index and get the list of documents {D1,D2} as an answer.

[1]Some of the information about inverted indexes in this section is from the Wikipedia pages "Search Engine Indexing" and "Inverted Index".

Word	**Documents**
is	D1, D2
john	D1
my	D1
name	D1, D2
what	D2
your	D2

Table 1: A simplified illustration of a sample inverted index where D1 has the content "my name is john" and D2 has "what is your name".

There are two main variants of inverted indexes: A ***record-level inverted index*** contains a list of references to documents for each word (as in the given example above). A ***word-level inverted index*** additionally contains the positions of each word within a document. The latter form offers more functionality (like phrase searches), but needs more processing power and space to be created.

3 Binary Search Tree Implementation (60 pts)

The binary search tree data structure used in this assignment is implemented as the class template `BST` with the template argument `T`, which is used as the type of the data stored in the nodes. The node of the binary search tree is implemented as the class template `BSTNode` with the template argument `T`, which is the type of the data stored in nodes. `BSTNode` class is the basic building block of the `BST` class. `BST` class has a single `BSTNode` pointer (namely `root`) in its private data field, which points to the root node of the binary search tree.

The `BST` class has its definition and implementation in *BST.hpp* file and the `BSTNode` class has its in *BSTNode.hpp* file.

3.1 `BSTNode`

`BSTNode` class represents nodes that constitute binary search trees. A `BSTNode` keeps two pointers (namely `left` and `right`) to the root nodes of its left and right subtrees, and a variable of type T (namely `data`) to hold the data. The class has two constructors and the overloaded output operator. They are already implemented for you. You should not change anything in file *BSTNode.hpp.*

3.2 `BST`

`BST` class implements a binary search tree data structure. `BST` class has a single data member in its private data field, which is a `BSTNode` pointer (namely `root`) that points to the root node of the binary search tree. There are some public member functions that are already implemented for you. You must provide implementations for the following public member functions that have been declared under indicated portions of *BST.hpp* file.

3.2.1 `BST();`

This is the default constructor. You should make necessary initializations in this function.

3.2.2 `BST(const BST<T> &obj);`

This is the copy constructor. You should make necessary initializations, create new nodes by copying the nodes in given `obj` and insert new nodes into the binary search tree. The structure among the nodes in given `obj` should also be copied to the new tree.

3.2.3 `~BST();`

This is the destructor. You should deallocate all memory that you were allocated before.

3.2.4 `BSTNode<T> *getRoot();`

This function should return a pointer to the root node in the binary search tree. If the binary search tree is empty, it should return `NULL`.

3.2.5 `void insert(const T &data);`

You should create a new node with given `data` and insert it to the appropriate location in the binary search tree. Do not forget to make necessary pointer modifications in the tree. You will not be asked to insert duplicated data to the binary search tree.

3.2.6 `void remove(const T &data);`

You should remove the node with given `data` from the binary search tree. Do not forget to make necessary pointer modifications in the tree. If there exists no such node in the binary search tree, do nothing. There will be no duplicated data in the binary search tree.

3.2.7 `BSTNode<T> *find(const T &data);`

You should search for the node in the binary search tree with the data same with the given `data` and return a pointer to that node. You can use the `operator==` to compare two `T` objects. If there exists no such node in the binary search tree, you should return `NULL`. There will be no duplicated data in the binary search tree.

3.2.8 `BST<T> &operator=(const BST<T> &rhs);`

This is the overloaded assignment operator. You should remove all nodes in the binary search tree and then create new nodes by copying the nodes in given `rhs` and insert new nodes into the binary search tree. The structure among the nodes in given `rhs` should also be copied to the new tree.

4 Inverted Index Implementation (40 pts)

Some details of the inverted indexes were given in section 2. For this assignment, you are expected to implement a ***word-level inverted index*** by using your binary search tree implementation. The inverted index in this assignment is implemented as the class `InvertedIndex`. `InvertedIndex` class has a `BST` object (namely `invertedIndex`) with the type `IIData` in its private data field. `IIData` class represents mappings in inverted indexes. It is the type of the data stored in nodes of the binary search tree.

The `InvertedIndex` class has its definition in *InvertedIndex.hpp* file and its implementation in *InvertedIndex.cpp* file. The `IIData` class has its definition in *IIData.hpp* file and its implementation in *IIData.cpp* file.

4.1 `IIData`

`IIData` class represents mappings that constitute inverted indexes. An `IIData` object keeps the `word` variable of type `std::string` and the `occurrences` variable of type `std::vector< std::pair< std::string, std::vector< int > > >` to hold the data related with the mapping. `occurrences` is the vector of existences of the word `word` in documents. Each occurrence is a pair of the form *(document_name, vector of positions (offsets))*. Let's suppose *"doc1.txt"* is *"my name is john"* and *"doc2.txt"* is *"what is your name name"*. Then the occurrences vector for the word *"name"* will be like *[(doc1.txt, [2]), (doc2.txt, [4, 5])]*.

Some public member functions have already been implemented for you. Do not change those implementations. In *IIData.cpp* file, you need to provide implementations for following functions declared under *IIData.hpp* header to complete the assignment. You should not change anything in file *IIData.hpp*.

4.1.1 `void addOccurrence(const std::string &documentName, int position);`

This is the member function to store existence (occurrence) of the word in a document to the inverted index. Name of the document and position (offset) of the word in the document are given as arguments. If the document is already in the occurrences vector, append the position to the document's vector of positons. If not, create a new pair and append it to the vector of occurrences.

4.1.2 `void removeOccurrences(const std::string &documentName);`

This is the member function to remove all existences of the word in a document from the inverted index. Name of the document is given as argument. You should remove the corresponding pair from the vector of occurrences.

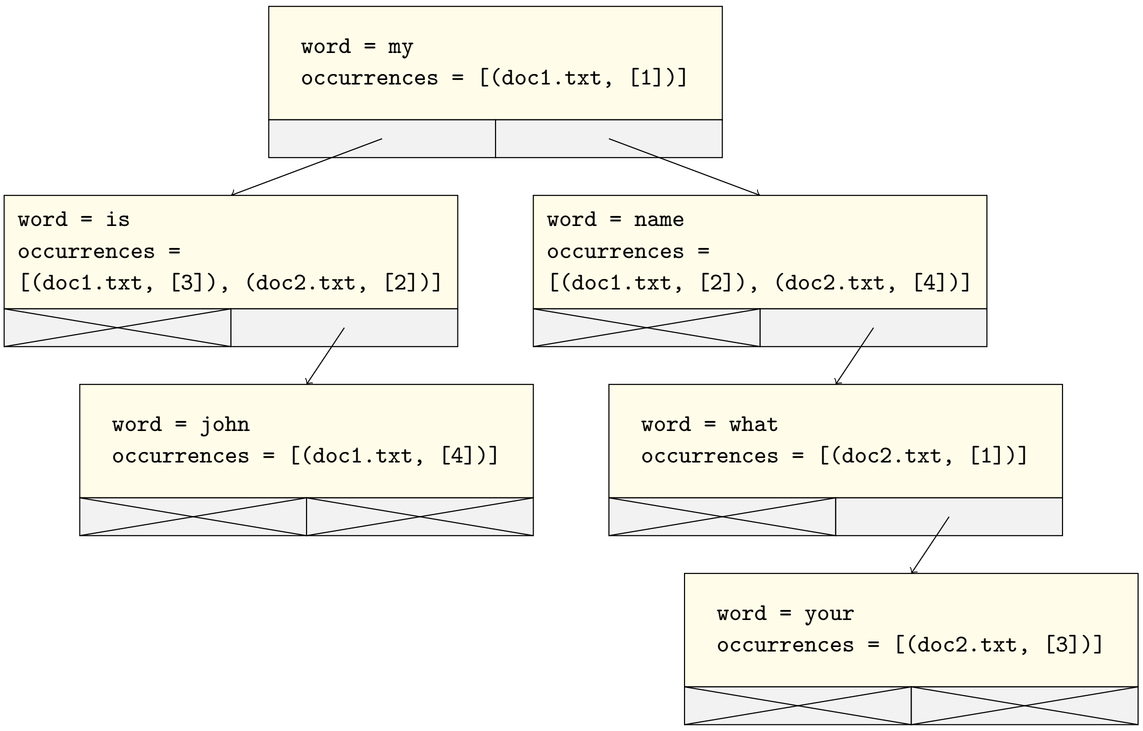
4.2 `InvertedIndex`

In `InvertedIndex` class, there is a single member variable named as `invertedIndex`, which is a `BST` object. Information of all mappings will be stored in this binary search tree and no other information will be stored. All member functions should utilize this binary search tree to operate as described in the following subsections. Some public member functions have already been implemented for you. Do not change those implementations. In *InvertedIndex.cpp* file, you need to provide implementations for following functions declared under *InvertedIndex.hpp* header to complete the assignment. You should not change anything in file *InvertedIndex.hpp*.

4.2.1 `InvertedIndex &addDocument(const std::string &documentName);`

This is the member function that takes the name of a document as string, reads that document word by word, and populates the inverted index with occurrences of words in the document. It is guaranteed that documents will be text files with extension ".txt" and their content will be list of words separated by single space characters. No characters other than [a-z] and the space character will be given in documents. Some parts of this function are already implemented for you. You should complete the implementation of this function.

For example, Figure 1 shows the resulting binary search tree representing the inverted index after adding two documents “doc1.txt” and “doc2.txt” respectively to the initially empty inverted index. In this scenario, “doc1.txt” is “my name is john” and “doc2.txt” is “what is your name”.



word = my
occurrences = [(doc1.txt, [1])]
word = is
occurrences =
[(doc1.txt, [3]), (doc2.txt, [2])]
word = name
occurrences =
[(doc1.txt, [2]), (doc2.txt, [4])]
word = john
occurrences = [(doc1.txt, [4])]
word = what
occurrences = [(doc2.txt, [1])]
word = your
occurrences = [(doc2.txt, [3])]

Figure 1: After adding two documents “doc1.txt” and “doc2.txt” respectively to the empty inverted index

4.2.2 `InvertedIndex &removeDocument(const std::string &documentName);`

This is the member function that takes the name of a document as string and removes all mappings related with that document from the inverted index. If there left no documents that are related with a word (e.g. no occurences lefts), then the node is completely removed from the binary search tree.

For example, Figure 2 shows the resulting binary search tree representing the inverted index after adding two documents “doc1.txt” and “doc2.txt” respectively to the initially empty inverted index and then removing the document “doc2.txt”. In this scenario, “doc1.txt” is “my name is john” and “doc2.txt” is “what is your name”. Note that the nodes in the binary search tree with word “what” and “your” are deleted.

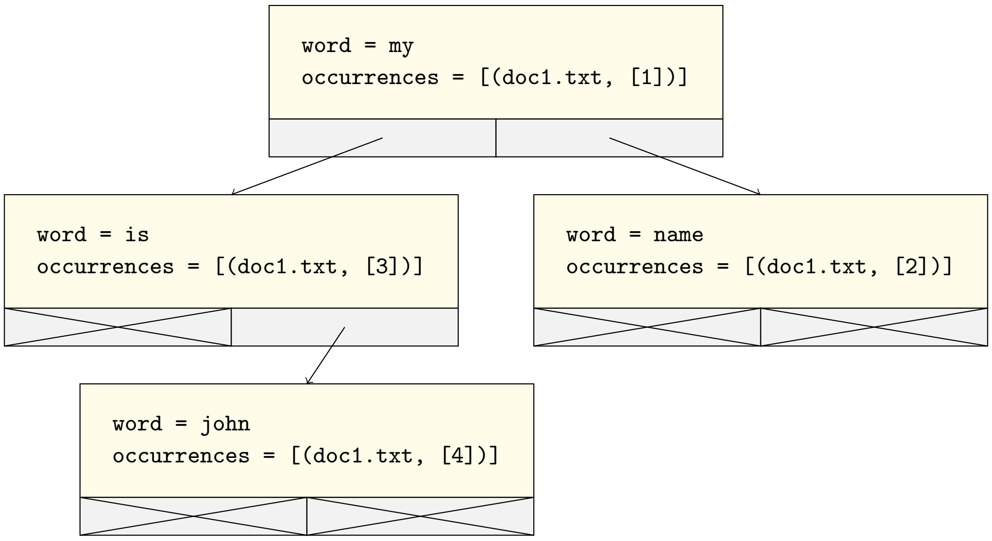

word = my
occurrences = [(doc1.txt, [1])]
word = is
occurrences = [(doc1.txt, [3])]
word = name
occurrences = [(doc1.txt, [2])]
word = john
occurrences = [(doc1.txt, [4])]

Figure 2: After adding two documents “doc1.txt” and “doc2.txt” respectively to the empty inverted index and then removing the document “doc2.txt”

5 Driver Programs

To enable you to test your binary search tree and inverted index implementations, two driver programs, *main_bst.cpp* and *main_invertedindex.cpp* are provided. Their expected outputs are also provided in *output_bst.txt* and *output_invertedindex.txt* files, respectively.

6 Regulations

1. **Programming Language:** You will use C++.
2. **S**tandard Template Library is allowed only for `vector` and `pair`. However, you should **not** use them for any purpose other than storing occurences of words in `IIData` class.
3. **E**xternal libraries other than those already included are **not** allowed.
4. **T**hose who do the operations (insert, remove, find, print) without utilizing the binary search tree will receive **0 grade**.
5. **T**hose who modify already implemented functions and those who insert other data variables or public functions and those who change the prototype of given functions will receive **0 grade**.
6. **T**hose who use compile-time arrays or variable-size arrays (not existing in ANSI C++) will receive **0 grade**. Options used for g++ are ”-ansi -Wall -pedantic-errors -O0”. They are already included in the provided Makefile.
7. **Y**ou can add private member functions whenever it is explicitly allowed.
8. **Late Submission:** Each assignment will have a fixed duration of 10 days and every student has a total of 5 days for late submissions during which no points will be deducted. Your assignment will not be accepted if you used up all of the 5 late days.

9. **Cheating:** We have zero tolerance policy for cheating. In case of cheating, all parts involved (source(s) and receiver(s)) get zero. People involved in cheating will be punished according to the university regulations. Remember that students of this course are bounded to code of honor and its violation is subject to severe punishment.

10. **Newsgroup:** You must follow the Forum (`odtuclass.metu.edu.tr`) for discussions and possible updates on a daily basis.

7 Submission

- Submission will be done via CengClass (`cengclass.ceng.metu.edu.tr`).
- Do not write a main function in any of your source files.
- A test environment will be ready in CengClass.
  - You can submit your source files to CengClass and test your work with a subset of evaluation inputs and outputs.
  - Additional test cases will be used for evaluation of your final grade. So, your actual grades may be different than the ones you get in CengClass.
  - Only the last submission before the deadline will be graded.